Chapter 4

Why Monads Matter

In this chapter we present a powerful device in state-of-the-art programming, that of a *monad*. The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, comprehension notation, state variable updating, context dependence, partial behaviour *etc.* in an elegant and uniform way.

Our motivation to this concept will start from a well-known problem in functional programming (and computing as a whole) — that of coping with undefined computations.

4.1 Partial functions

Consider the $\mathbb{R}$ to $\mathbb{R}$ function

$$g\,x \overset{\rm def}{=} 1/x$$

Clearly, g is undefined for $x = 0$ because $g\,0 = 1/0$ is so big a real number that it cannot be properly evaluated. In fact, the HASKELL output for $g\,0 = 1/0$ is just "panic":

```
Main> g 0

Program error: {primDivDouble 1.0 0.0}

Main>
```

Functions such as g above are called *partial functions* because they cannot be applied to all of their inputs (*i.e.*, they diverge for some of their inputs). Partial functions are very common in mathematics or programming — for other examples think of *e.g.* list-processing functions `head` and `tail`.

Panic is very dangerous in programming. In order to avoid this kind of behaviour one has two alternatives, either ensuring that every call to $g\,x$ is *protected* — *i.e.*, the contexts which wrap up such calls ensure *pre-condition* $x \neq 0$, or one *raises* exceptions, *i.e.* explicit error values. In the former case, mathematical proofs need to be carried out in order to ensure *safety* (that is, *pre-condition* compliance). The overall effect is that of *restricting* the domain of the partial function. In the latter case one goes the other way round, by extending the co-domain (vulg. range) of the function so that it accommodates exceptional outputs. In this way one might define, in HASKELL:

```
data ExtReal = Ok Real | Error
```

and then redefine

```
g :: Real -> ExtReal
g 0 = Error
g n = Ok 1/n
```

In general, one might define parametric type

```
data Ext a = Ok a | Error
```

in order to extend an arbitrary data type `a` with its (polymorphic) exception (or error value). Clearly, the isomorphisms hold:

$$\mathsf{Ext}\,A \cong \mathsf{Maybe}\,A \cong 1 + A$$

So, in abstract terms, one may regard as *partial* every function of type

$$1 + A \xleftarrow{\;g\;} B$$

for some A and B [1].

4.2 Putting partial functions together

Do partial functions compose? Their types won't match in general:

$$\begin{tikzcd}
 & 1 + B \arrow[d, dotted, no head] & A \arrow[l, "g"'] \\
1 + C & B \arrow[l, "f"'] &
\end{tikzcd}$$

[1]In conventional programming, every function delivering a *pointer* as result — as in *e.g.* the C programming language — can be regarded as one of these functions.

Clearly, we have to extend f — which is itself a partial function — to some f' able to accept arguments from $1 + B$:

$$\begin{tikzcd}
 & 1 \arrow[d, "i_1"] \arrow[dl, "\cdots"'] & \\
 & 1+B \arrow[dl, "f'"] & A \arrow[l, "g"'] \\
1+C & B \arrow[l, "f"] \arrow[u, "i_2"'] &
\end{tikzcd}$$

The most "obvious" instance of the ellipsis $(\ldots)$ in the diagram above is i_1 and this corresponds to what is called *strict* composition: an exception produced by the *producer* function g is propagated to the output of the *consumer* function f:

$$f \bullet g \overset{\text{def}}{=} [\, i_1, f \,] \cdot g \tag{4.1}$$

Expressed in terms of `Ext`, composite function $f \bullet g$ works as follows:

$$(f \bullet g)a = f'(g\,a)$$

where

$$\begin{aligned} f'\,\texttt{Error} &= \texttt{Error} \\ f'\,(\texttt{Ok}\,b) &= f\,b \end{aligned}$$

Altogether, we have the following Haskell pointwise expression for $f \bullet g$:

```
\a -> f' (g a)
      where f' Error = Error
            f' (Ok b) = f b
```

Note that the adopted extension of f can be decomposed — by reverse $+$-absorption (2.41) — into

$$f' = [\, i_1, id \,] \cdot (id + f)$$

as displayed in diagram

$$\begin{tikzcd}
1+(1+C) \arrow[d, "{[\, i_1, id \,]}"'] & 1+B \arrow[l, "id+f"'] \arrow[d, dash, dotted] & A \arrow[l, "g"'] \\
1+C & B \arrow[l, "f"'] &
\end{tikzcd}$$

All in all, we have the following version of (4.1):

$$f \bullet g \overset{\text{def}}{=} [\, i_1, id \,] \cdot (id + f) \cdot g$$

Does this functional composition scheme have a unit, that is, is there a u such that

$$f \bullet u = f = u \bullet f \tag{4.2}$$

holds? Clearly, if it exists, it must bear type $1 + A \xleftarrow{u} A$. Let us *solve* (4.2) for u:

$$\begin{array}{ll}
& f \bullet u = f = u \bullet f \\
\equiv & \qquad \{ \text{ substitution } \} \\
& [\, i_1, f \,] \cdot u = f = [\, i_1, u \,] \cdot f \\
\Leftarrow & \qquad \{ \text{ let } u = i_2 \ \} \\
& [\, i_1, f \,] \cdot i_2 = f = [\, i_1, i_2 \,] \cdot f \wedge u = i_2 \\
\equiv & \qquad \{ \text{ by } +\text{-cancellation (2.38) and } +\text{-reflection (2.39) } \} \\
& f = f = id \cdot f \wedge u = i_2 \\
\Leftarrow & \qquad \{ \text{ identity } \} \\
& u = i_2
\end{array}$$

4.3 Lists

In contrast to partial functions, which may produce *no* output, let us now consider functions which deliver *too many* outputs, for instance, lists of output values:

$$\begin{tikzcd}
 & B^\star \arrow[d, dotted, no head] & A \arrow[l, "g"'] \\
C^\star & B \arrow[l, "f"'] &
\end{tikzcd}$$

Functions f and g do not compose but, once again, one can think of extending the consumer function (f) by mapping it along the output of the producer function (g):

$$\begin{tikzcd}
(C^\star)^\star \arrow[d, dotted, no head] & B^\star \arrow[l, "f^\star"'] \arrow[d, dotted, no head] \\
C^\star & B \arrow[l, "f"']
\end{tikzcd}$$

To complete the process, one has to *flatten* the nested-sequence output in $(C^\star)^\star$ via the obvious list-catamorphism $C^\star \xleftarrow{concat} (C^\star)^\star$, where $concat \stackrel{\mathrm{def}}{=} (\![\, [\,], +\!\!+ \,]\!)$. In summary:

$$f \bullet g \stackrel{\mathrm{def}}{=} concat \cdot f^\star \cdot g \tag{4.3}$$

as captured in the following diagram:

$$\begin{tikzcd}
(C^\star)^\star \arrow[d, "concat"'] & B^\star \arrow[l, "f^\star"'] \arrow[d, dotted, no head] & A \arrow[l, "g"'] \\
C^\star & B \arrow[l, "f"'] &
\end{tikzcd}$$

Exercise 4.1. *Show that* singl *(recall exercise 3.7) is the unit* u *of* $\bullet$ *in the context of (4.3).*
□

Exercise 4.2. *Encode in* HASKELL *a pointwise version of (4.3).* ***Hint***: *first apply (list) cata-absorption (3.67).*
□

4.4 Monads

Both function composition schemes (4.1) and (4.3) above share the same polytypic pattern: the output of the producer function is “F-times” more elaborate than the input of the consumer function, where F is some parametric datatype — $\mathsf{F}\,X = 1+X$ in case of (4.1), and $\mathsf{F}\,X = X^\star$ in case of (4.3). Then a composition scheme is devised for such functions, which is displayed in

$$\begin{tikzcd}
\mathsf{F}(\mathsf{F}\,C) \arrow[d, "\mu"'] & \mathsf{F}\,B \arrow[l, "\mathsf{F}\,f"'] \arrow[d, dotted, no head] & A \arrow[l, "g"'] \\
\mathsf{F}\,C & B \arrow[l, "f"'] &
\end{tikzcd}$$

and is given by

$$f \bullet g \overset{\text{def}}{=} \mu \cdot \mathsf{F}\, f \cdot g \tag{4.4}$$

where $\mathsf{F}\, A \xleftarrow{\mu} \mathsf{F}^2\, A$ is a suitable polymorphic function. (In the case of $\mu = [\, i_1, id\,]$ in case (4.1), and $\mu = concat$ in case (4.3).)

Together with a unit function $\mathsf{F}\, A \xleftarrow{u} A$ and μ, datatype F will form a so-called *monad* type, of which $(1+)$ and $(\_)^\star$ are the two examples seen above. Arrow $\mu \cdot \mathsf{F}\, f$ is called the *extension* of f. Functions μ and u are referred to as the monad's *multiplication* and *unit*, respectively. The monadic composition scheme (4.4) is referred to as *Kleisli composition*.

A *monadic arrow* $\mathsf{F}B \xleftarrow{f} A$ conveys the idea of a function which produces an output of "type" B "wrapped by F", where datatype F describes some kind of (computational) "effect". The monad's unit $\mathsf{F}B \xleftarrow{u} B$ is a primitive monadic arrow which produces (*i.e.* promotes, injects, wraps) data *together with* such an effect.

The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, state variable updating, context dependence, partial behaviour (seen above) *etc.* in an elegant and uniform way. Moreover, the monad's operators exhibit notable properties which make it possible to *reason* about such computational effects.

The remainder of this section is devoted to such properties. First of all, the properties implicit in the following diagrams will be *required* for F to be regarded as a monad:

Multiplication :

$$\begin{tikzcd} \mathsf{F}^2\, A \arrow[d, "\mu"'] & \mathsf{F}^3\, A \arrow[l, "\mu"'] \arrow[d, "\mathsf{F}\,\mu"] \\ \mathsf{F}\, A & \mathsf{F}^2\, A \arrow[l, "\mu"] \end{tikzcd}$$

$$\mu \cdot \mu = \mu \cdot \mathsf{F}\, \mu \tag{4.5}$$

Unit :

$$\begin{tikzcd} \mathsf{F}^2\, A \arrow[d, "\mu"'] & \mathsf{F}\, A \arrow[l, "u"'] \arrow[d, "\mathsf{F}\,u"] \arrow[dl, "id"] \\ \mathsf{F}\, A & \mathsf{F}^2\, A \arrow[l, "\mu"] \end{tikzcd}$$

$$\mu \cdot u = \mu \cdot \mathsf{F}\, u = id \tag{4.6}$$

Simple but beautiful symmetries apparent in these diagrams make it easy to memorize their laws and check them for particular cases. For instance, for the $(1+)$ monad, law (4.6) will read as follows:

$$[\, i_1, id\,] \cdot i_2 = [\, i_1, id\,] \cdot (id + i_2) = id$$

These equalities are easy to check.

In laws (4.5) and (4.6), the different instances of μ and u are differently typed, as these are polymorphic and exhibit natural properties:

μ-natural :

$$\begin{tikzcd} A \arrow[d, "f"'] & \mathsf{F}\,A & \mathsf{F}^2\,A \arrow[l, "\mu"'] \arrow[d, "\mathsf{F}^2\,f"] \\ B & \mathsf{F}\,B \arrow[from=u, "\mathsf{F}\,f"'] & \mathsf{F}^2\,B \arrow[l, "\mu"] \end{tikzcd} \qquad \mathsf{F}\,f \cdot \mu = \mu \cdot \mathsf{F}^2\,f \tag{4.7}$$

u-natural :

$$\begin{tikzcd} A \arrow[d, "f"'] & \mathsf{F}\,A & A \arrow[l, "u"'] \arrow[d, "f"] \\ B & \mathsf{F}\,B \arrow[from=u, "\mathsf{F}\,f"'] & B \arrow[l, "u"] \end{tikzcd} \qquad \mathsf{F}\,f \cdot u = u \cdot f \tag{4.8}$$

The simplest of all monads is the *identity monad* $\mathsf{F}\,X \stackrel{\rm def}{=} X$, which is such that $\mu = id$, $u = id$ and $f \bullet g = f \cdot g$. So — in a sense — the whole functional discipline studied thus far was already *monadic*, over the simplest of all monads: the identity one. Put in other words, such functional discipline can be framed into a wider discipline in which an arbitrary monad is present — the one we study in this chapter.

4.4.1 Properties involving (Kleisli) composition

The following properties arise from the definitions and monadic properties presented above:

$$\begin{aligned} f \bullet (g \bullet h) &= (f \bullet g) \bullet h && (4.9) \\ u \bullet f = \; & f \; = f \bullet u && (4.10) \\ (f \bullet g) \cdot h &= f \bullet (g \cdot h) && (4.11) \\ (f \cdot g) \bullet h &= f \bullet (\mathsf{F}\,g \cdot h) && (4.12) \\ id \bullet id &= \mu && (4.13) \end{aligned}$$

Properties (4.9) and (4.10) are the monadic counterparts of, respectively, (2.8) and (2.10), meaning that monadic composition preserves the properties of normal functional composition. In fact, for the identity monad, these properties coincide with each other.

Above we have shown that property (4.10) holds for the list monad, recall (4.2). A general proof can be produced similarly. We select property (4.9) as an illustration of the rôle of the monadic properties:

$$
\begin{array}{cl}
 & f \bullet (g \bullet h) \\
= & \quad \{ \text{ definition (4.4) twice } \} \\
 & \mu \cdot \mathsf{F}\, f \cdot (\mu \cdot \mathsf{F}\, g \cdot h) \\
= & \quad \{ \; \mu \text{ is natural (4.7) } \} \\
 & \mu \cdot \mu \cdot \mathsf{F}(\mathsf{F}\, f) \cdot \mathsf{F}\, g \cdot h \\
= & \quad \{ \text{ functor } \mathsf{F} \; \} \\
 & \mu \cdot \mu \cdot \mathsf{F}(\mathsf{F}\, f \cdot g) \cdot h \\
= & \quad \{ \text{ definition (4.4) } \} \\
 & \mu \cdot (\mathsf{F}\, f \cdot g) \bullet h \\
= & \quad \{ \text{ definition (4.4) } \} \\
 & (f \bullet g) \bullet h
\end{array}
$$

Exercise 4.3. *Check the other laws above.*
□

4.5 Monadic application (binding)

The monadic extension of functional application ap (2.67) is another operator ap' which is intended to be "tolerant" in face of any F'ed argument x:

$$
\begin{array}{c}
(\mathsf{F}\, B)^A \times \mathsf{F}\, A \xrightarrow{ap'} \mathsf{F}\, B \\
ap'(f, x) = f'\, x = (\mu \cdot \mathsf{F}\, f)x
\end{array}
\tag{4.14}
$$

If in curry/flipped format, monadic application is called *binding* and denoted by symbol "$\gg\!=$", looking very much like postfix functional application,

$$
((\mathsf{F}\, B)^A)^{\mathsf{F}\, A} \xrightarrow{\gg=} \mathsf{F}\, B \tag{4.15}
$$

that is:

$$x \gg\!= f \overset{\text{def}}{=} (\mu \cdot \mathsf{F}\, f)x \tag{4.16}$$

This operator will exhibit properties arising from its definition and the basic monadic properties, *e.g.*

$$\begin{array}{cl} & x \gg\!= u \\ \equiv & \quad \{ \text{ definition (4.16) } \} \\ & (\mu \cdot \mathsf{F}\, u)x \\ \equiv & \quad \{ \text{ law (4.6) } \} \\ & (id)x \\ \equiv & \quad \{ \text{ identity function } \} \\ & x \end{array}$$

At pointwise level, one may chain monadic compositions from left to right, *e.g.*

$$(((x \gg\!= f_1) \gg\!= f_2) \gg\!= \ldots f_{n-1}) \gg\!= f_n$$

for functions $A \xrightarrow{f_1} \mathsf{F}\, B_1$, $B_1 \xrightarrow{f_2} \mathsf{F}\, B_2$, ... $B_{n-1} \xrightarrow{f_n} \mathsf{F}\, B_n$.

4.6 Sequencing and the do-notation

Given two monadic values x and y, it becomes possible to "sequence" them, thus obtaining another of such value, by defining the following operator:

$$x \gg y \overset{\text{def}}{=} x \gg\!= \underline{y} \tag{4.17}$$

For instance, within the finite-list monad, one has

$$[1,2] \gg [3,4] = (concat \cdot \underline{[3,4]}^{\star})[1,2] = concat[[3,4],[3,4]] = [3,4,3,4]$$

Because this operator is associative (prove this as an exercise), one may iterate it to more than two arguments and write, for instance,

$$x_1 \gg x_2 \gg \ldots \gg x_n$$

This leads to the popular do notation, which is another piece of (pointwise) notation which makes sense in a monadic context:

$$\mathsf{do}\ x_1; x_2; \ldots; x_n \overset{\text{def}}{=} x_1 \gg \mathsf{do}\ x_2; \ldots; x_n$$

for $n \geq 1$. For $n = 1$ one trivially has

$$\mathsf{do}\ x_1 \overset{\text{def}}{=} x_1$$

4.7 Generators and comprehensions

The do-notation accepts a variant in which the arguments of the $\gg$ operator are "generators" of the form

$$a \leftarrow x \tag{4.18}$$

where, for a of type A, x is an inhabitant of monadic type $\mathsf{F}\,A$. One may regard $a \leftarrow x$ as meaning "let a be taken from x". Then the do-notation extends as follows:

$$\mathsf{do}\ a \leftarrow x_1; x_2; \ldots; x_n \overset{\text{def}}{=} x_1 \ggg\!= \lambda a.(\mathsf{do}\ x_2; \ldots; x_n) \tag{4.19}$$

Of course, we should now allow for the x_i to range over terms involving variable a. For instance, by writing (again in the list-monad)

$$\mathsf{do}\ a \leftarrow [1,2,3]; [a^2] \tag{4.20}$$

we mean

$$\begin{array}{ll} & [1,2,3] \ggg\!= \lambda a.[a^2] \\ = & \mathit{concat}((\lambda a.[a^2])^\star [1,2,3]) \\ = & \mathit{concat}[[1],[4],[9]] \\ = & [1,4,9] \end{array}$$

The analogy with classical set-theoretic ZF-notation, whereby one might write $\{a^2 \mid a \in \{1,2,3\}\}$ to describe the set of the first three perfect squares, calls for the following notation,

$$[\, a^2 \mid a \leftarrow [1,2,3]\,] \tag{4.21}$$

as a "shorthand" of (4.20). This is an instance of the so-called *comprehension* notation, which can be defined in general as follows:

$$[\, e \mid a_1 \leftarrow x_1, \ldots, a_n \leftarrow x_n \,] = \mathsf{do}\ a_1 \leftarrow x_1; \ldots; a_n \leftarrow x_n; u(e) \tag{4.22}$$

where u is the monad's unit (4.6,4.8).

Alternatively, comprehensions can be defined as follows, where p, q stand for arbitrary generators:

$$[t] = u\,t \tag{4.23}$$

$$[\, f\,x \mid x \leftarrow l\,] = (\mathsf{F}\,f)l \tag{4.24}$$

$$[\, t \mid p, q\,] = \mu[\,[\,t \mid q\,] \mid p\,] \tag{4.25}$$

Note, however, that comprehensions are not restricted to lists or sets — they can be defined for any monad F.

4.8 Monads in HASKELL

In the *Standard Prelude* for HASKELL, one finds the following minimal definition of the `Monad` class,

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
```

where `return` refers to the unit of `m`, on top of which the "sequence" operator

```
    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a
```

is defined by

```
    p >> q  = p >>= \ _ -> q
```

as expected. This class is instantiated for finite sequences (`[]`), `Maybe` and `IO`.

The μ multiplication operator is function `join` in module `Monad.hs`:

```
    join :: (Monad m) => m (m a) -> m a
    join x = x >>= id
```

This is easily justified:

$$\begin{aligned} join\, x &= x \ggg\!= id \qquad (4.26)\\ &= \quad \{\ \text{definition (4.16)}\ \}\\ &\quad (\mu \cdot \mathsf{F}\, id)x\\ &= \quad \{\ \text{functors commute with identity (3.44)}\ \}\\ &\quad (\mu \cdot id)x\\ &= \quad \{\ \text{law (2.10)}\ \}\\ &\quad \mu\, x \end{aligned}$$

In `Mpi.hs` we define (Kleisli) monadic composition in terms of the binding operator:

```
(.!) :: Monad a => (b -> a c) -> (d -> a b) -> d -> a c
(f .! g) a = (g a) >>= f
```

4.8.1 Monadic I/O

IO, a parametric datatype whose inhabitants are special values called *actions* or *commands*, is a most relevant monad. Actions perform the interconnection between HASKELL and the environment (file system, operating system). For instance, $getLine :: \mathsf{IO}\, String$ is a particular action. Parameter `String` refers to the fact that this action "delivers" — or extracts — a string from the environent. This meaning is clearly conveyed by the type `String` assigned to symbol l in

$$\texttt{do}\ \ l \leftarrow getLine; \ldots l \ldots$$

which is consistent with typing rule for generators (4.18). Sequencing corresponds to the ";" syntax in most programming languages (*e.g.* C) and the `do`-notation is particulary intuitive in the IO-context.

Examples of functions delivering actions are

$$FilePath \xrightarrow{readFile} \mathsf{IO}\, String$$

and

$$Char \xrightarrow{putChar} \mathsf{IO}()$$

— both produce I/O commands as result.

As is to be expected, the implementation of the IO monad in HASKELL — available from the *Standard Prelude* — is not totally visible, for it is bound to deal with the intrincacies of the underlying machine:

```
instance Monad IO where
    (>>=)  = primbindIO
    return = primretIO
```

Rather interesting is the way IO is regarded as a functor:

```
    fmap f x = x >>= (return . f)
```

This goes the other way round, the monadic structure "helping" in defining the functor structure, everything consistent with the underlying theory:

$$\begin{array}{rcl}
x \gg\!= (u \cdot f) & = & (\mu \cdot \mathsf{IO}(u \cdot f))x \\
& = & \qquad \{ \text{ functors commute with composition } \} \\
& & (\mu \cdot \mathsf{IO}\, u \cdot \mathsf{IO}\, f)x \\
& = & \qquad \{ \text{ law (4.6) for } \mathsf{F} = \mathsf{IO} \ \} \\
& & (\mathsf{IO}\, f)x \\
& = & \qquad \{ \text{ definition of } fmap \ \} \\
& & (fmap\, f)x
\end{array}$$

For enjoyable reading on monadic input/output in HASKELL see [Hud00], chapter 18.

Exercise 4.4. *Use the* `do`*-notation and the comprehension notation to output the following truth-table, in* HASKELL*:*

p / q	*False*	*True*
False	*False*	*False*
True	*False*	*True*

□

Exercise 4.5. *Extend the* Maybe *monad to the following "error message" exception handling datatype:*

```
data Error a = Err String | Ok a deriving Show
```

In case of several error messages issued in a `do` *sequence, how many turn up on the screen? Which ones?*
□

4.9 The state monad

NB: this section is still a draft

The so-called *state monad* is a monad whose inhabitants are state-transitions encoding a particular brand of state-based automaton known as *Mealy machine*. Given a set A (input alphabet), a set B (output alphabet) and a set of states S, a deterministic Mealy machine (DMM) is specified by a transition function of type

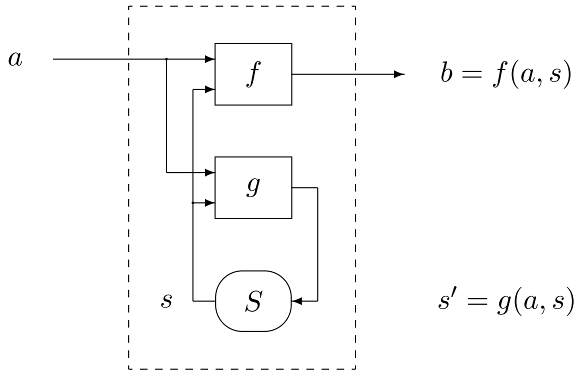
$$A \times S \xrightarrow{\delta} B \times S \tag{4.27}$$

Wherever $(b, s') = \delta(a, s)$, we say that the machine has transition

$$s \xrightarrow{a|b} s'$$

and refer to s as the **before** state, and to s' as the **after** state.

It is clear from (4.27) that δ can be expressed as the *split* of two functions f and g, $\delta = \langle f, g\rangle$, as depicted in the following diagram:



(4.28)

The information recorded in the state of a DMM is either meaningless to the user of the machine (as in eg. the case of states represented by numbers) or too complex to be perceived and handled explicitly (as is the case of eg. the data kept in a large database). So, it is convenient to *abstract* from it. Such an abstraction leads to the *state monad* in the following way: taking (4.27) and recalling (2.75), we simply transpose (ie. *curry*) δ and obtain

$$A \xrightarrow{\overline{\delta}} \underbrace{(B \times S)^S}_{(\mathsf{St}\ S)\ B} \tag{4.29}$$

thus "shifting" the input state to the output. In this way, $\overline{\delta}\ a$ is a function capturing all state-transitions (and corresponding outputs) for input a. For instance, the function which *appends* a new element to the back of a queue,

$$enq(a,s) \stackrel{\text{def}}{=} s + [a]$$

can be converted into a DMM by adding to it a dummy output of type 1 and then transposing:

$$\begin{array}{rcl} enqueue & : & A \rightarrow (1 \times S)^S \\ enqueue\ a & \stackrel{\text{def}}{=} & \langle !, (+[a])\rangle \end{array} \tag{4.30}$$

Action $enqueue$ performs enq on the state while acknowledging it by issuing an output of type 1.

Unit and multiplication. Let us show that

$$(\mathsf{St}\ S)\ A \quad \cong \quad (A \times S)^S \tag{4.31}$$

forms a monad. As we shall see, the fact that the *values* of this monad are functions brings the theory of exponentiation to the forefront. Thus, a review of section 2.14 is recommended at this point.

Notation $\widehat{f}$ will be used to abbreviate $uncurry\ f$, enabling the following variant of universal law (2.67),

$$\widehat{k} = f \quad \Leftrightarrow \quad f = ap \cdot (k \times id) \tag{4.32}$$

whose cancellation

$$\widehat{k} = ap \cdot (k \times id) \tag{4.33}$$

is written pointwise as follows:

$$\widehat{k}(c, a) \quad = \quad (k\ c)a \tag{4.34}$$

First of all, what is the functor behind (4.31)? Fixing the state space S, we obtain

$$\mathsf{F}X \quad \stackrel{\mathrm{def}}{=} \quad (X \times S)^S \tag{4.35}$$

on objects and

$$\mathsf{F}f \quad \stackrel{\mathrm{def}}{=} \quad (f \times id)^S \tag{4.36}$$

on functions, where $(\_)^S$ is the exponential functor (2.71).

The unit of this monad is the transpose of the simplest of all Mealy machines — the identity:

$$\begin{array}{rcl} u & : & A \to (A \times S)^S \\ u & = & \overline{id} \end{array} \tag{4.37}$$

Let us see what this means:

$$\begin{array}{cl} & u = \overline{id} \\ \equiv & \qquad \{\ (2.67)\ \} \\ & ap \cdot (u \times id) = id \\ \equiv & \qquad \{\ \text{introducing variables}\ \} \\ & ap(u\ a, s) = (a, s) \\ \equiv & \qquad \{\ \text{definition of } ap\ \} \\ & (u\ a)s = (a, s) \end{array}$$

So, action $u\ a$ performed on state s keeps s unchanged and outputs a.

From the type of μ, for this monad,

$$((A \times S)^S \times S)^S \xrightarrow{\mu} (A \times S)^S$$

one figures out $\mu = x^S$ (recalling the exponential functor as defined by (2.71)) for x of type $((A \times S)^S \times S) \xrightarrow{x} (A \times S)$. This, on its turn, is easily recognized as an instance of the ap polymorphic function (2.67), which is such that $ap = \widehat{id}$, recall (2.69). Altogether, we define

$$\mu \quad = \quad ap^S \tag{4.38}$$

Let us inspect the behaviour of μ by checking the meaning of applying it to an action expressed as in diagram (2.75):

$$\begin{array}{cl}
& \mu\langle f, g\rangle = ap^S\langle f, g\rangle \\
\equiv & \qquad \{ \text{ (2.71) } \} \\
& \mu\langle f, g\rangle = ap \cdot \langle f, g\rangle \\
\equiv & \qquad \{ \text{ extensional equality (2.5) } \} \\
& \mu\langle f, g\rangle s = ap(f\ s, g\ s) \\
\equiv & \qquad \{ \text{ definition of } ap \ \} \\
& \mu\langle f, g\rangle s = (f\ s)(g\ s)
\end{array}$$

We find out that μ "unnests" the action inside f by applying it to the state delivered by g.

Checking the monadic laws. The calculation of (4.6) is made in two parts, checking $\mu \cdot u = id$ first,

$$\begin{array}{cl}
& \mu \cdot u \\
= & \qquad \{ \text{ definitions } \} \\
& ap^S \cdot \overline{id} \\
= & \qquad \{ \text{ exponentials absorption (2.72) } \} \\
& \overline{ap \cdot id} \\
= & \qquad \{ \text{ reflection (2.69) } \} \\
& id
\end{array}$$

and then checking $\mu \cdot (\mathsf{F}u) = id$:

$$
\begin{array}{ll}
& \mu \cdot (\mathsf{F}u) \\
= & \qquad \{ \text{ (4.38,4.36) } \} \\
& ap^S \cdot (\overline{id} \times id)^S \\
= & \qquad \{ \text{ functor } \} \\
& (ap \cdot (\overline{id} \times id))^S \\
= & \qquad \{ \text{ cancellation (2.68) } \} \\
& id^S \\
= & \qquad \{ \text{ functor } \} \\
& id
\end{array}
$$

The proof of (4.5) is also not difficult once supported by the laws of exponentials.

Kleisli composition. Let us calculate $f \bullet g$ for this monad:

$$
\begin{array}{ll}
& f \bullet g \\
= & \qquad \{ \text{ (4.4) } \} \\
& \mu \cdot \mathsf{F}\, f \cdot g \\
= & \qquad \{ \text{ (4.38) ; (4.36) } \} \\
& ap^S \cdot (f \times id)^S \cdot g \\
= & \qquad \{ \text{ } (\_)^S \text{ is a functor } \} \\
& (ap \cdot (f \times id))^S \cdot g \\
= & \qquad \{ \text{ (4.32) } \} \\
& \widehat{f}^S \cdot g \\
= & \qquad \{ \text{ cancellation } \} \\
& \widehat{f}^S \cdot \overline{\widehat{g}} \\
= & \qquad \{ \text{ absorption (2.72) } \} \\
& \overline{\widehat{f} \cdot \widehat{g}}
\end{array}
$$

In summary, we have:

$$
f \bullet g \;=\; \overline{\widehat{f} \cdot \widehat{g}} \tag{4.39}
$$

which can be written alternatively as

$$\widehat{f \bullet g} \;=\; \widehat{f} \cdot \widehat{g}$$

Let us use this in calculating law

$$pop \bullet push \;=\; u \tag{4.40}$$

where $push$ and pop are such that

$$\begin{array}{rcl} push & : & A \to (1 \times S)^S \\ \widehat{push} & \stackrel{\rm def}{=} & \langle !, \widehat{(:)} \rangle \end{array} \tag{4.41}$$

$$\begin{array}{rcl} pop & : & 1 \to (A \times S)^S \\ \widehat{pop} & \stackrel{\rm def}{=} & \langle head, tail \rangle \cdot \pi_2 \end{array} \tag{4.42}$$

for S the datatype of finite lists. We reason:

$$\begin{array}{cl} & pop \bullet push \\ = & \quad \{\ (4.39)\ \} \\ & \overline{\widehat{pop} \cdot \widehat{push}} \\ = & \quad \{\ (4.41, 4.42)\ \} \\ & \overline{\langle head, tail \rangle \cdot \pi_2 \cdot \langle !, \widehat{(:)} \rangle} \\ = & \quad \{\ (2.20, 2.24)\ \} \\ & \overline{\langle head, tail \rangle \cdot \widehat{(:)}} \\ = & \quad \{\ out \cdot in = id \text{ (lists)}\ \} \\ & \overline{id} \\ = & \quad \{\ (4.37)\ \} \\ & u \end{array}$$

Bind. The effect of binding a state transition x to a state-monadic function h is calculated in a similar way:

$$\begin{array}{cl} & x \gg\!= h \\ = & \quad \{\ (4.16)\ \} \\ & (\mu \cdot \mathsf{F}h)x \end{array}$$

$$
\begin{array}{ll}
= & \{\ (4.38) \text{ and } (4.36)\ \} \\
& (ap^S \cdot (h \times id)^S)x \\
= & \{\ (\_)^S \text{ is a functor }\ \} \\
& (ap \cdot (h \times id))^S x \\
= & \{\ \text{cancellation } (4.33)\ \} \\
& \widehat{h}^S x \\
= & \{\ \text{exponential functor } (2.71)\ \} \\
& \widehat{h} \cdot x
\end{array}
$$

Let us unfold $\widehat{h} \cdot x$ by splitting x into its components two components f and g:

$$
\begin{array}{ll}
& \langle f, g\rangle \gg\!= h \quad = \quad \widehat{h} \cdot \langle f, g\rangle \\
\equiv & \{\ \text{go pointwise }\ \} \\
& (\langle f, g\rangle \gg\!= h)s \quad = \quad \widehat{h}(\langle f, g\rangle s) \\
\equiv & \{\ (2.18)\ \} \\
& (\langle f, g\rangle \gg\!= h)s \quad = \quad \widehat{h}(f\ s, g\ s) \\
\equiv & \{\ (4.34)\ \} \\
& (\langle f, g\rangle \gg\!= h)s \quad = \quad h(f\ s)(g\ s)
\end{array}
$$

In summary, for a given "before state" s, $g\ s$ is the intermediate state upon which $f\ s$ runs and yields the output and (final) "after state".

Two prototypical inhabitants of the state monad: get and put. These generic actions are defined as follows, in the PF-style:

$$get \stackrel{\text{def}}{=} \langle id, id\rangle \tag{4.43}$$

$$put \stackrel{\text{def}}{=} \overline{\langle !, \pi_1\rangle} \tag{4.44}$$

Action g retrieves the data stored in the state without changing it, while put — which can also be written

$$put\ s \quad = \quad \langle !, \underline{s}\rangle \tag{4.45}$$

or even as

$$put\ s \quad = \quad modify\ \underline{s} \tag{4.46}$$

where

$$modify\ f \overset{\text{def}}{=} \langle !, f \rangle \tag{4.47}$$

updates the state via state-to-state function f — stores a particular value in the state.

The following is an example, in Haskell, of the standard use of get/put in managing context data, in this case a counter. The function decorates each node of a $BTree$ (recall this datatype from page 91) with its position in the tree:

```
decBTree Empty = return Empty
decBTree (Node (a,(t1,t2))) =
        do  n <- get ;
            put(n+1) ;
            x <- decBTree t1 ;
            y <- decBTree t2 ;
            return (Node((a,n),(x,y)))
```

To close the chapter, we will present a strategy for deriving this kind of monadic functions.

4.10 'Monadification' of Haskell code made easy

There is an easy roadmap for "monadification" of Haskell code. What do we mean by *monadification*? Well, in a sense — as we shall soon see — every piece of code is monadic: we don't notice this because the underlying monad is *invisible*. We are going to see how to make it visible taking advantage of monadic `do` notation and leaving it open for instantiation. This will bridge the gap between monads' theory and its application to handling particular effects in concrete situations.

Let us take as starting point the pointwise version of sum, the list catamorphism which adds all numbers found in its input:

```
sum [] = 0
sum (h:t) = h + sum t
```

Notice that this code could have been written as follows

```
sum [] = id 0
sum (h:t) = let x = sum t in id (h+x)
```

using let notation and two instances of the identity function. Question: why such a "baroque" version of the starting, so simple a piece of code? Answer:

- The let ... in ... notation stresses the fact that recursive call happens *earlier* than the delivery of the result.

- The id functions signal the exit points of the algorithm, that is, the points where it *returns* something to the caller.

Next, let us

- re-write *id* into `return`;
- re-write `let x = ... in ...` into `do { x <- ... ; ... }`

One will obtain the following version of sum:

```
msum [] = return 0
msum (h:t) = do {x <- msum t ; return (h+x) }
```

Typewise, while sum has type `(Num a) => [a] -> a`, $msum$ has type

```
(Monad m, Num a) => [a] -> m a
```

That is, $msum$ is monadic — parametric on monad m — while sum is not.

There is a particular monad for which sum and $msum$ coincide: the **identity** monad $\mathsf{Id}\ X = X$. It is very easy to show that inside this monad `return` is the identity and `do` means the same as `let` — enough for the pointwise versions of the two functions to be the same. Thus the "invisible" monad mentioned earlier is the identity monad.

In summary, the monadic version is *generic* in the sense that it runs on whatever monad you like, enabling you to perform *side effects* while the code runs. If you don't need any effects then you get the "non-monadic" version as special case, as seen above. Otherwise, Haskell will automatically switch to the effects you want, depending on the monad you choose (often determined by context).

For each particular monad we may decide to add specific monadic code like `get` and `put` in the `decBTree` example, where we want to take advantage of the state monad. As another example, check the following enrichment of $msum$ with state-monadic code helping you to trace the execution of your program:

```
msum' [] = return 0
msum' (h:t) =
     do {x <- msum' t ;
         print ("x= " ++ show x);
         return (h+x) }
```

Thus one obtains traces the code in the way prescribed by the particular usage of the $print$ (state monadic) function:

```
*Main> msum' [3,5,1,3,4]
"x= 0"
```

```
"x= 4"
"x= 7"
"x= 8"
"x= 13"
*Main>
```

In the reverse direction, one may try and see what happens to monadic code upon removing all monad-specific functions and going into the identity monad once it gets monad generic. In the case of `decBTree`, for instance, we will get

```
decBTree Empty = return Empty
decBTree (Node (a,(t1,t2))) =
        do
            x <- decBTree t1 ;
            y <- decBTree t2 ;
            return (Node(a,(x,y)))
```

once `get` and `put` are removed (and therefore all instances of `n`), and then

```
decBTree Empty = Empty
decBTree (Node (a,(t1,t2))) =
        let
            x =  decBTree t1
            y =  decBTree t2
        in  Node(a,(x,y))
```

This is the identity function on type BTree, recall the cata-reflection law (3.58). So, the *archetype* of (inspiration for) much monadic code is the most basic of all tree traversal functions — the identity [2]. The same could be said about imperative code of a particular class — the *recursive descent* one — much used in construction, for instance.

Playing with effects

As it may seem from the previous examples, adding effects to produce monadic code is far from arbitrary. This can be further appreciated by defining the function that yields the smallest element of a list,

```
getmin [a] = a
getmin (h:t) = min h (getmin t)
```

[2]We have seen the same kind of "inspiration" before in building type functors (3.66) which, for $f = id$, boil down to the identity.

which is incomplete in the sense that it does not specify the meaning of `getmin []`. As this is mathematically undefined, it should be expressed "outside the maths", that is, as an effect. Thus, to complete the defintion we first go monadic, as we did before,

```
mgetmin [a] = return a
mgetmin (h:t) = do {x <- mgetmin t ; return (min h x) }
```

and then chose a monad in which to express the meaning of `getmin []`, for instance the `Maybe` monad

```
mgetmin [] = Nothing
mgetmin [a] = return a
mgetmin (h:t) = do {x <- mgetmin t ; return (min h x) }
```

Alternatively, we might have written

```
mgetmin [] = Error "Empty input"
```

going into the `Error` monad, or even the simpler (yet interesting) `mgetmin [] = []`, which shifts the code into the list monad, yielding singleton lists in the success case, otherwise the empty list.

Function `getmin` above is an example of a partial function, that is, a function which is undefined for some of its inputs. These functions cause much interference in functional programming, which monads help us to keep under control.

Let us see how such interference is coped with in the case of higher order functions, taking `map` as example

```
map f [] = []
map f (h:t) = (f h): map f t
```

and supposing `f` is not a total function. How do we cope with erring evaluations of `f h`? As before, we first "letify" the code,

```
map f [] = []
map f (h:t) = let
      b = f h
      x = map f t in b:x
```

we go monadic in the usual way,

```
mmap f [] = return []
mmap f (h:t) = do { b <- f h ; x <- mmap f t ; return (b:x) }
```

and everything goes smoothly — as can be checked, the function thus built is of the expected (monadic) type:

```
mmap :: (Monad m) => (a -> m b) -> [a] -> m [b]
```

Run `mmap Just [1,2,3,4]`, for instance: you will obtain `Just [1,2,3,4]`. Now run `mmap print [1,2,3,4]`. You will see the items in the sequence printed sequentially.

One may wonder about the behaviour of the `mmap` for `f` the identity function: will we get an error? No, we get a well-typed function of type `[m a] -> m [a]`, for `m` a monad. We thus obtain the well-known monadic function `sequence` which evaluates each *action* in the input sequence, from left to right, collecting the results. For instance, applying this function to input sequence `[Just 1, Nothing, Just 2]` the output will be `Nothing`.

There is much more one could say about monadic programming, but this is enough to see *where such a programming style comes from*.

Exercise 4.6. *Use the* monadification *technique to encode monadic function*

```
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

which generalizes the list-based filter function.
□

Exercise 4.7. *"Reverse" the following monadic code into its non-monadic archetype:*

```
f :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
f p [] = return []
f p (h:t) = do { b <- p h ;
                 t' <- f p t;
                 return(if b then h:t' else [])
               }
```

Which function of the Haskell Prelude do you get by such reverse monadification*?*
□

To be continued

4.11 Bibliography notes

The use of monads in computer science started with Moggi [Mog89], who had the idea that monads should supply the extra semantic information needed to implement the lambda-calculus theory. Haskell [Jon03] is among the computer languages which make systematic use of monads for implementing effects and imperative constructs in an otherwise purely functional language.

Category theorists invented monads in the 1960's to concisely express certain aspects of universal algebra. Functional programmers invented list comprehensions in the 1970's to concisely express certain programs involving lists. Philip Wadler [Wad89] made a great contribution in the field by showing that list comprehensions could be generalised to arbitrary monads and unify with imperative "do"-notation in case of the monad which explains imperative computations.

Monads are nowadays an essential feature of functional programming and are used in fields as diverse as language parsing [HM93], component-oriented programming [Bar01], strategic programming [LV03] and multimedia [Hud00].